The University of Melbourne
School of Computing and Information Systems
SWEN20003 Object Oriented Software Development

ShadowDance

Project 1, Semester 2, 2023
Released: Friday, 25th August 2023 at 4:30pm AEST
Initial Submission Due: Thursday, 31st August 2023 at 4:30pm AEST
Project Due: Friday, 8th September 2023 at 4:30pm AEST

Please read the complete specification before starting on the project, because there are important instructions through to the end!

Overview

Welcome to the first project for SWEN20003, Semester 2, 2023. Across Project 1 and 2, you will design and create a music arcade game in Java called *ShadowDance* (an adaptation of the original 90's arcade game *Dance Dance Revolution*). In this project, you will create the first level of the full game that you will complete in Project 2B. This is an **individual project**. You may discuss it with other students, but all of the implementation must be your **own work**. By submitting the project you declare that you understand the University's policy on academic integrity and aware of consequences of any infringement, including the use of **artificial intelligence**.

You may use any platform and tools you wish to develop the game, but we recommend using IntelliJ IDEA for Java development as this is what we will support in class.

The purpose of this project is to:

- Give you experience working with an object-oriented programming language (Java),
- Introduce simple game programming concepts (2D graphics, input, simple calculations)
- Give you experience working with a simple external library (Bagel)

Extensions & late submissions: If you need an extension for the project, please complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

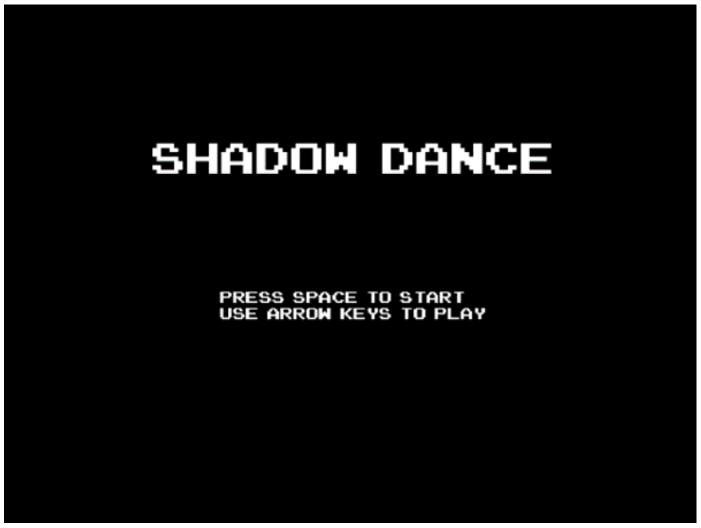
If you submit late (**either** with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions. All of this is explained again in more detail at the end of this specification.

You **must** make at least 5 commits (excluding the Initial Submission commit) throughout the development of the project, and they must have meaningful messages. This is also explained in more detail at the end of the specification.

Game Overview

*"The aim is simple : the **player** has to hit the corresponding musical **notes** that appear on screen in different **lanes** on time to score **points**. Can you beat the target score to win the game?"*

The game consists of three levels - Project 1 will only feature the first level. The notes will descend from the top vertically in the 4 lanes. The player has to press the corresponding arrow key when the note overlaps with the stationary note symbol at the bottom. The accuracy of how close the note was to the stationary note when the key was pressed, will determine the points given. There will be special **hold notes** that require the player to hold down the key. To win, the player needs to beat the target score when all the notes have fallen. If the player's score is lower, the game ends.



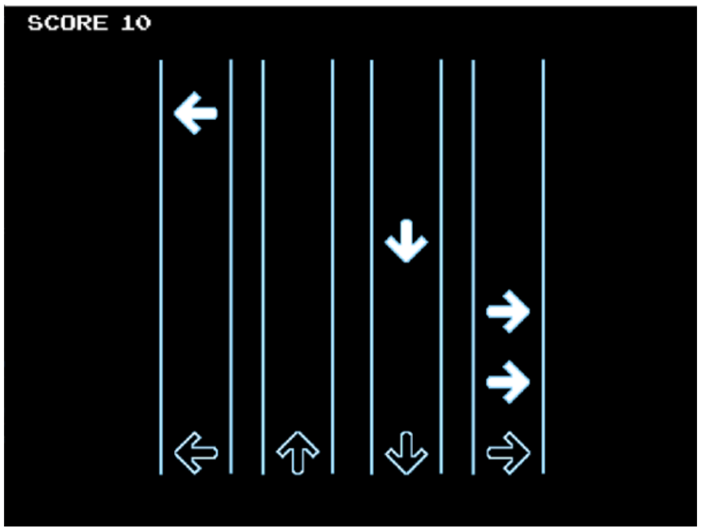


Figure 1: Completed Project 1 Screenshot

The Game Engine

The **B**asic **A**cademic **G**ame **E**ngine **L**ibrary (Bagel) is a game engine that you will use to develop your game. You can find the documentation for Bagel here.

Coordinates

Every coordinate on the screen is described by an (x, y) pair. $(0, 0)$ represents the top-left of the screen, and coordinates increase towards the bottom-right. Each of these coordinates is called a *pixel*. The Bagel `Point` class encapsulates this.

Frames

Bagel will refresh the program's logic at the same refresh rate as your monitor. Each time, the screen will be cleared to a blank state and all of the graphics are drawn again. Each of these steps is called a **frame**. Every time a frame is to be rendered, the `update()` method in `ShadowDance` is called. It is in this method that you are expected to update the state of the game.

The refresh rate is now typically `120` times per second (Hz) but some devices might have a lower rate of 60Hz. In this case, when your game is running, it may look different to the demo videos as the constant values in this specification have been chosen for a refresh rate of 120Hz. For your convenience, when writing and testing your code, you **may** change these values to make your game playable (these changes are explained later). If you do change the values, **remember** to change them back to the original specification values before submitting, as your code will be **marked on 120Hz screens.**

The Game Elements

Below is an outline of the different game elements you will need to implement.

Window and Background

The background (`background.png`) should be rendered on the screen and completely fill up your window throughout the game. The default window size should be `1024 * 768` pixels. The background has already been implemented for you in the skeleton package.

Messages

All messages should be rendered with the font provided in the `res` folder (`FSO8BITR.ttf`), in size `64` (unless otherwise specified). All messages should be roughly centered both horizontally and vertically (unless otherwise specified).

Hint: The `drawString()` method in the Font class uses the given coordinates as the bottom left of the message. So to center the message, you will need to calculate the coordinates using the `Window.getWidth(), Window.getHeight() and Font.getWidth()` methods, and also the font size.

Game Start

When the game is run, a title message that reads `SHADOW DANCE` should be rendered in the font provided. The bottom left corner of this message should be located at `(220, 250)`.

Additionally, an instruction message consisting of 2 lines:

```
PRESS SPACE TO START
USE ARROW KEYS TO PLAY
```

should be rendered **below** the title message, in the font provided, in size `24`. The bottom left of the first line in the message should be calculated as follows: the x-coordinate should be increased by `100` pixels and the y-coordinate by `190` pixels.

There must be **adequate spacing** between the 2 lines to ensure readability (you can decide on the value of this spacing yourself, as long as it's not small enough that the text overlaps or too big that it doesn't fit within the screen).

To help when testing your game, you can allow the player to pause the game (i.e. everything in the window will stop moving) by pressing the `Tab` key (this is **not assessed** but will help you when coding!)

World File

The lanes and the notes will be defined in a **world file**, describing the type and their position or time of appearance in the window. The world file is located at `res/level1.csv`. (For 60Hz screens, use the provided `res/level1-60.csv` file which has all the values halved). A world file is a comma-separated value (CSV) file with rows in one of the following formats:

```
Lane, type of lane, x-coordinate
            (or)
Type of lane, type of note, frame-number
```

An example of a world file:

```
Lane,Left,282
Lane,Right,742
Left,Normal,398
Down,Hold,1262
```

The **type of lane** refers to the arrow key which it corresponds to, for example: the first entry in the above example is for the lane corresponding to the left arrow key. The **type of note** refers to whether the note is normal or a hold note, and the **frame number** is the frame in which the note starts appearing on screen. For example, the third entry in the above example refers to a left arrow key note of the normal type that appears from the 199th frame onwards.

You must actually load it—copying and pasting the data, for example, is not allowed. You have been provided with an extra world file to test on, located at `res/test1.csv`. Marking will be conducted on a hidden **different** CSV file of the same format. **Note:** For Project 1, you can assume that there will always be four lanes, and at least one note for each of the four directions.

Win Conditions

The player gains or loses points based on how close the note was to the stationary note when the correct arrow key is pressed (how this is calculated is explained later). When all the notes in the CSV file have fallen, if the player's score is higher than the target score of **150**, this is considered as a win. A winning message that reads `CLEAR!` should be rendered as described earlier in the *Messages* section.

Lose Conditions

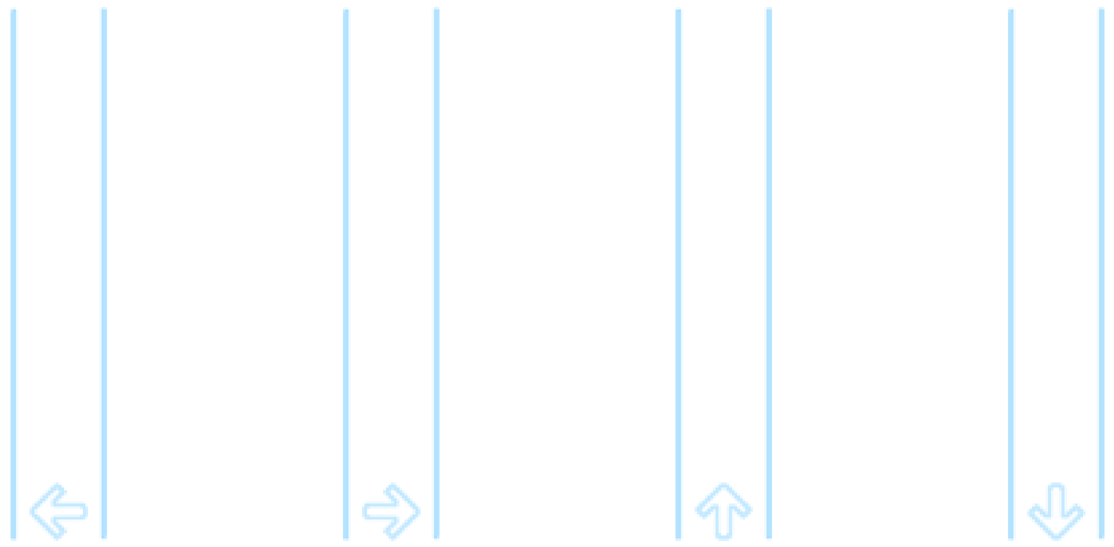
The game will continue running until all the notes have fallen. After this, the game will end if the player's score is less than the target score of **150**. A message of `TRY AGAIN` should be rendered as described earlier in the *Messages* section. If the player terminates the game window at any point (by pressing the Escape key or by clicking the Exit button), the window will simply close and no message will be shown.

Game Entities

The following game entities have an associated image (or multiple!) and a starting location `(x, y)`. Remember that all images are drawn from the centre of the image using these coordinates.

Lane

In this game, there are **four** lanes for the four arrow keys (left, right, up and down) represented by the images shown on the next page. The `(x, y)` coordinate of the centre of each image is as follows : the x-coordinate is given in the CSV file and the y-coordinate is **384**. The position of the lane stays constant throughout the game.

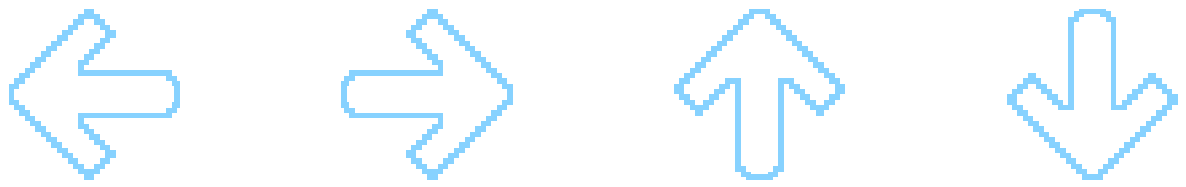


(a) laneLeft.png (b) laneRight.png (c) laneUp.png (d) laneDown.png

Figure 2: The lane images (note that the size has been reduced to fit on the page)

A lane can have a maximum of `100` notes and `20` hold notes. Each lane has a stationary note symbol at the bottom which is the target. The player needs to press the corresponding arrow key when the falling note overlaps with the stationary note to score points. The y-coordinate of the centre of the four stationary notes is **657**.

Note



(a) noteLeft.png (b) noteRight.png (c) noteUp.png (d) noteDown.png

Figure 3: The note images

There are **four** types of notes for the four arrow keys represented by the above images. Each note will descend vertically from the top of the screen in the corresponding lane, for example: a left note should descend in the left lane and an up note should descend in the up lane. The starting `(x, y)` coordinate of the centre of each note image is as follows : the x-coordinate is the **same** as the x-coordinate of the lane it corresponds to and the y-coordinate is **100**.

Each note moves downwards at a speed of **2 pixels per frame** (for 60Hz screens, increase this value to 4). The frame number from which the note starts being drawn on screen is given in the CSV file. The note will continue to be drawn until either it either leaves the window from the bottom of the screen or the player presses the corresponding key for the note (and a score is calculated as described below).

Note Scoring

The score for each key press is calculated based on the **accuracy** of how close the given note was to the stationary note symbol in the lane when the key was pressed. When the key is pressed, the **absolute distance** in pixels between the y-coordinate of the centre of the falling note and the y-coordinate of the centre of the stationary note is calculated. The method to determine the score from this distance is shown below.

- If distance <= 15, this is a `PERFECT` score and receives `10` points
- If 15 < distance <= 50, this is a `GOOD` score and receives `5` points
- If 50 < distance <= 100, this is a `BAD` score and receives `-1` points
- If 100 < distance <= 200, this is a `MISS` and receives `-5` points.

If the note leaves the window from the bottom of the screen without the corresponding key being pressed, this is considered as a `MISS` too and receives `-5` points.

When a score is calculated, the corresponding **score message** (shown in the list above) must be rendered on screen as described in the *Messages* section. The font size must be set to `40` and the message must be rendered for `30` frames. For example, when the score is perfect, the text rendered must be `PERFECT`.

The player's current **total score** must also be rendered on screen. The score is rendered in the top left corner of the screen in the format of `"SCORE k"` where `k` is the current score. The bottom left corner of this message should be located at `(35, 35)` and the font size should be 30.

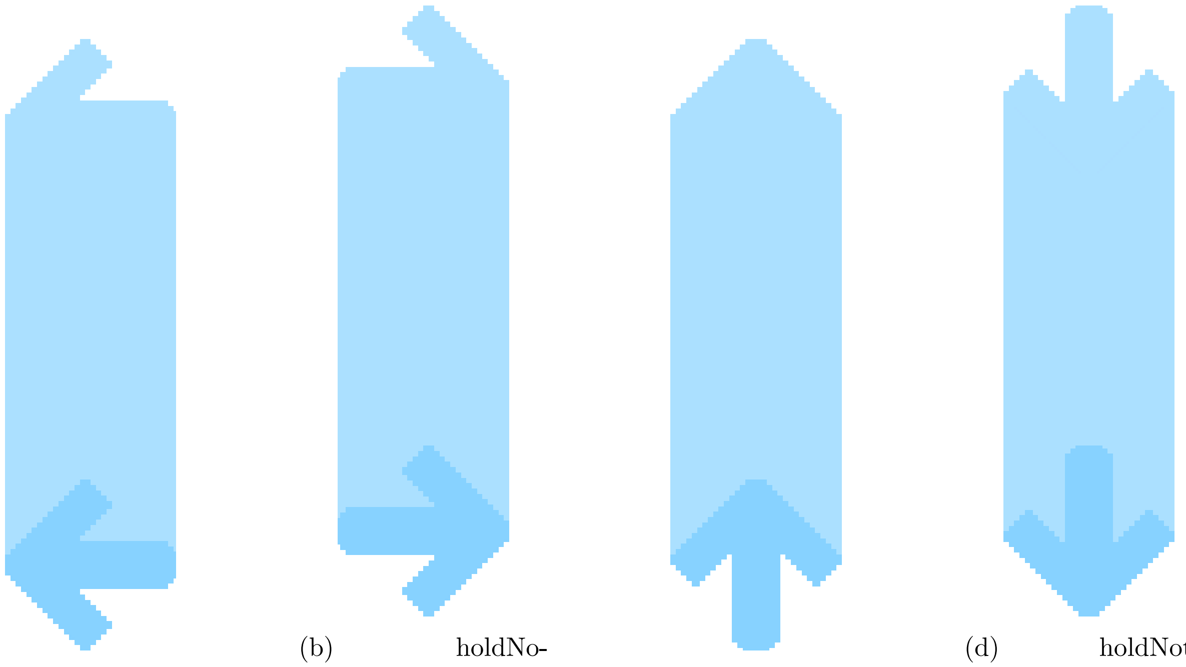
SCORE 9

Figure 4: Score

Hold Note

A hold note is a special type of note, where the player has to **hold down** the corresponding arrow key for the duration in which the falling hold note overlaps with the stationary note in the lane. Once again, there are **four** types of hold notes for the four arrow keys shown in the figure below.

The starting `(x, y)` coordinate of the centre of each image is as follows : the x-coordinate is the **same** as the x-coordinate of the lane it corresponds to and the y-coordinate is **24**. The speed is the **same** as for a normal note and the frame number is also given in the CSV file.



(a) holdNoteLeft.png (b) holdNoteRight.png (c) holdNoteUp.png (d) holdNoteDown.png

Figure 5: The hold note images

Hold Note Scoring

For hold notes, the scoring is calculated in a similar manner but the difference is that **two** scores are calculated - first when the hold is **started** and the second when the hold is **released**. **Hint:** To check if a hold has been released, you will find the `wasReleased()` method in the `Input` class helpful.

When the hold starts, the y-coordinate at the **bottom** of the image of the holding note needs to be used to calculate the distance (instead of the centre of the image like in a normal note). Likewise when the hold is released, the y-coordinate at the **top** of the image needs to be used. **Hint:** To calculate the y-coordinate at the bottom of the hold note image, add `82` to the centre y-coordinate, and for the y-coordinate at the top of the image, subtract `82`.

The method of determining the score from the distance is the same as given in the *Note Scoring* section (bottom of page 6). If the hold note leaves the window from the bottom of the screen without the hold starting, this is considered as a `MISS` and receives `-5` points. If the hold was released at a distance greater than `200` pixels, this is a `MISS` too.

Similar to normal notes, the score messages should also be rendered on screen and the scores from hold notes, need to be added to the total score too.

Sound

Note that this section is optional and you will not be assessed on this.

This game wouldn't be a correct representation of Dance Dance Revolution without any music *right?* You will be given a separate file called `musicDemo.zip` which contains a demo using the `javax` package, on how to add sound to your project. The code has a class called `Track` which encapsulates all the logic related to creating, playing and pausing a sound. If you want to add sound, use this class in your project. The code currently uses a demo sound track - you may add any sound you want as long as the track is a .wav file. You can add more than one sound, by creating multiple `Track` objects and passing the file paths for the sounds in the constructor. (Note that if you add several sounds, it may cause gameplay delays on computers with low resources). We hope you have fun with this!

Your Code

You must submit a class called `ShadowDance` that contains a `main` method that runs the game as prescribed above. You may choose to create as many additional classes as you see fit, keeping in mind the principles of object oriented design discussed so far in the subject. You will be assessed based on your code running correctly, as well as the effective use of Java concepts. As always in software engineering, appropriate comments and variables/method/class names are important.

Implementation Checklist

To get you started, here is a checklist of the game features, with a suggested order for implementing them:

- Draw the title and game instruction messages on screen
- Draw the lanes on screen
- Read the world file, and draw the notes on screen
- Implement movement logic for the notes
- Implement the scoring behaviour when a key is pressed for a note
- Draw the corresponding scoring messages
- Implement the same behaviour as above for the hold notes
- Implement win detection and draw winning message on screen
- Implement lose detection and draw losing message on screen

Supplied Package

You will be given a package called `project-1-skeleton.zip` that contains the following: (1) Skeleton code for the `ShadowDance` class to help you get started, stored in the `src` folder. (2) All graphics and fonts that you need to build the game, stored in the `res` folder. (3). The `pom.xml` file required for Maven. Here is a more detailed description:

- `res/` – The graphics and font for the game (You are not allowed to modify any of the files in this folder).
  - `background.png`: The image to represent the background.
  - `laneLeft.png`: The image to represent a lane for the left arrow key.
  - `laneRight.png`: The image to represent a lane for the right arrow key.
  - `laneUp.png`: The image to represent a lane for the up arrow key.
  - `laneDown.png`: The image to represent a lane for the down arrow key.
  - `noteLeft.png`: The image to represent a note for the left arrow key.
  - `noteRight.png`: The image to represent a note for the right arrow key.
  - `noteUp.png`: The image to represent a note for the up arrow key.
  - `noteDown.png`: The image to represent a note for the down arrow key.
  - `holdNoteLeft.png`: The image to represent a hold note for the left arrow key.
  - `holdNoteRight.png`: The image to represent a hold note for the right arrow key.
  - `holdNoteUp.png`: The image to represent a hold note for the up arrow key.
  - `holdNoteDown.png`: The image to represent a hold note for the down arrow key.
  - `FSO8BITR.ttf`: The font to be used throughout this game.
  - `level1.csv`: The world file for the first level.
  - `level1-60.csv`: Same file as above but for 60Hz screens.
  - `test1.csv`: A test file for you to try.
  - `test1-60.csv`: Same file as above but for 60Hz screens.
  - `track1.wav`: Sample music track which is the same as the file in musicDemo.zip (you can ignore this file if you're not adding music).
  - `resources.txt`: The file containing credit for the font and music (you can ignore this file).
- `src/` – The skeleton code for the game.
  - `ShadowDance.java`: The skeleton code that contains entry point to the Game, a `readCSV()` method and an `update()` method that draws the background.
- `pom.xml`: File required to set up Maven dependencies.

Submission and Marking

Initial Submission

To ensure you start the project with a correct set-up of your local and remote repository, you must complete this Initial Submission procedure on or before **Thursday, 31st August 2023 at 4:30pm**.

1. Clone the `[user-name]-project-1` folder from GitLab.
2. Download the `project-1-skeleton.zip` package from LMS, under Project 1.
3. Unzip it.
4. Move the **contents** of the unzipped folder to the `[user-name]-project-1` folder **in your local machine**.
5. Add, commit and push this change to your remote repository with the commit message `"initial submission"`.
6. Check that your push to Gitlab was successful and to the correct place.

After completing this, you can start implementing the project by adding code to meet the requirements of this specification. Please remember to add, commit and push your code regularly with meaningful commit messages as you progress.

You **must** complete the Initial Submission following the above instructions by the due date. Not doing this will incur a **penalty of 3 marks** for the project. It is best to do the Initial Submission before starting your project, so you can make regular commits and push to Gitlab since the very start. However, if you start working on your project locally before completing Initial Submission, that is fine too, just make sure you move all of the contents from your project folder to `[user-name]-project-1` in your local machine.

Technical requirements

- The program must be written in the Java programming language.
- Comments and class names must be in English **only**.
- The program must not depend upon any libraries other than the Java standard library and the Bagel library (as well as Bagel's dependencies).
- The program must compile fully without errors.

Submission will take place through GitLab. You are to submit to your `<username>-project-1` repository. At the **bare minimum** you are expected to follow the structure below. You **can** create more files/directories in your repository if you want.

```
username-project-1
├─ res
│  └─ resources used for project 1
└─ src
   ├─ ShadowDance.java
   └─ other Java files
```

On 8th September 2023 at 4:30pm, your latest commit will automatically be harvested from GitLab.

Commits

You are free to push to your repository post-deadline, but only the latest commit on or before 8th September 2023 4:30pm will be marked. You **must** make at least 5 commits (excluding the Initial Submission commit) throughout the development of the project, and they must have meaningful messages (commit messages must match the code in the commit). If commits are anomalous (e.g. commit message does not match the code, commits with a large amount of code within two commits which are not far apart in time) you risk penalization.

Examples of **good, meaningful** commit messages:

- implemented movement logic
- fix the note's scoring behaviour
- refactored code for cleaner design

Examples of **bad, unhelpful** commit messages:

- fesjakhbdjl
- yeah easy finished the logic
- fixed thingzZZZ

Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should not go back and comment your code after the fact. You should try to comment as you go. *(Yes, we can tell).*
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private (apart from constants).
- Any constant should be defined as a final static variable (**Note:** for Image and Font objects, use only final). Constants can be public depending on usage. Don't use magic numbers!
- Think about whether your code is written to be easily extensible via appropriate use of classes.

- Make sure each class makes sense as a cohesive whole. A class should have a single well-defined purpose, and should contain all the data it needs to fulfil this purpose.

Extensions and late submissions

If you need an **extension** for the project, please complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

The project is due at **4:30pm sharp**. Any submissions received past this time (from 4:30pm onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit **late** (*either* with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions.

Marks

Project 1 is worth **10** marks out of the total 100 for the subject. Although you may see how inheritance can be used for future extensibility, you are **not required** to use inheritance in this project.

- **NOTE:** Not completing the Initial Submission (described in the Submission and Marking section here) before beginning your project will result in a **3 mark penalty**!
- Features implemented correctly – **6 marks**
  - Starting screen is implemented correctly: (**0.5 marks**)
  - The lanes' images are implemented correctly: (**0.5 marks**)
  - The notes' images and movement (including hold notes) are implemented correctly: (**1.5 marks**)
  - The scoring of notes (including hold notes) is implemented correctly: (**1.5 marks**)
  - Player's total score is implemented correctly: (**1 mark**)
  - Win detection is implemented correctly with winning message rendered: (**0.5 marks**)
  - Loss detection is implemented correctly with game-over message: (**0.5 marks**)
- Code (coding style, documentation, good object-oriented principles) – **4 marks**
  - Delegation and Cohesion – breaking the code down into appropriate classes, each being a complete unit that contain all their data: (**1.5 marks**)
  - Use of Methods – avoiding repeated code and overly long/complex methods: (**1 mark**)
  - Code Style – visibility modifiers, consistent indentation, lack of magic numbers, commenting, etc. : (**1.5 marks**)